Lektion 4

In dieser Lektion werden zunächst zwei Problemstellungen betrachtet, die typischerweise rekursiv gelöst werden: Die Berechnung aller Teilmengen einer Menge (als Stellvertreter für typische kombinatorische Probleme) und die systematische Suche nach Lösungen mittels sog. Backtracking.

Danach steigen wir ein in das folgende Kapitel zur Analyse von Algorithmen. Dabei geht es um die Frage der Korrektheit und der Effizienz von Algorithmen.

3.3 Rekursive Berechnung der Potenzmenge

In diesem Abschnitt betrachten wir eine nichttriviale Problemstellung aus der Kombinatorik, die ohne Rekursion nicht so leicht zu lösen wäre.

Beispiel 3.13 - Potenzmenge berechnen

- Gegeben ist ein Array mit n unterschiedlichen Werten (= Menge M).
- Berechne die Menge aller Teilmengen $\wp(M)$ von M (die *Potenzmenge* der Menge).

Die Potenzmenge ist eine Menge von Mengen. In der Potenzmenge $\wp(M)$ sind immer die leere Menge $\varnothing$ sowie die komplette Menge M als Teilmengen enthalten.

Beispiel:

- Menge: $M = \{a,b,c\}$
- Potenzmenge: $\wp(M) = \{ \varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\} \}$

Rekursiver Lösungsansatz

Wie können wir die 2^n Teilmengen berechnen? Um auf eine rekursive Lösung zu kommen, gehen wir wieder wie üblich vor:

a) In welchen einfachen Fällen kann die Lösung direkt angegeben werden?

   Der einfachste Fall ist sicher die leere Menge, d.h. M = $\varnothing$.
   Die leere Menge hat nur sich selbst als Teilmenge, d.h. $\wp(\varnothing) = \{ \varnothing \}$

b) Wie können in nicht einfachen Fällen, d.h. die Menge hat mindestens ein Element, die Teilmengen bestimmt werden?

Sei Menge $M = \{a_1, ..., a_{n-1}, a_n\}$ nicht leer ($n \geq 1$).

1. Wir wählen ein Element der Menge, z.B. a_n.
2. Wenn wir nun alle Teilmengen von M betrachten, dann gibt es zwei Arten von Teilmengen:

   T^+ Menge aller Teilmengen, die a_n enthalten

   T^- Menge aller Teilmengen, die a_n nicht enthalten

   Die Menge aller Teilmengen von M ist dann die Vereinigung beider Sammlungen von Teilmengen, d.h.

   $$\wp(M) = T^- \cup T^+$$

Wie lassen sich die Sammlungen T^- und T^+ berechnen? Hier kommt nun Rekursion ins Spiel.

3. Bei T^- können wir das Problem auf ein kleineres, gleichartiges Problem zurückführen:
   Die Menge aller Teilmengen von $\{a_1, ..., a_{n-1}, a_n\}$, die a_n nicht enthalten, ist nichts anderes als die Menge aller Teilmengen von $M\backslash\{a_n\} = \{a_1, ..., a_{n-1}\}$, d.h. einer Menge mit einem Element weniger.

   $$T^- = \wp(M\backslash\{a_n\})$$

   Wir haben somit ein kleineres, gleichartiges Problem. Wir können die Menge der Teilmengen rekursiv berechnen.
4. Wie kommt man nun zu T^+, d.h. der Menge aller Teilmengen, die das gewählte Element a_n enthalten? Auch das ist leicht zu bewerkstelligen: Wenn man eine Teilmenge hat, die a_n enthält und entfernt a_n daraus, dann hat man eine Teilmenge der „Restmenge“ $\{a_1, ..., a_{n-1}\}$. Umgekehrt heißt das, nimmt man die Teilmengen aus T^- (alle ohne a_n) und nimmt zu jeder dieser Teilmengen jeweils a_n dazu, dann erhält man genau alle Teilmengen aus T^+, d.h. alle, die a_n enthalten.

Beispiel: Wenn M = {a, b, c}

Wähle z.B. c als Element aus:

T^-: alle Teilmengen ohne c, also alle Teilmengen von {a, b}:

$$T^- = \{ \varnothing,\ \{a\},\ \{b\},\ \{a,b\} \}$$

T^+: alle Teilmengen mit c: Nimm zu jeder Teilmenge aus T^- das Element c dazu.

$$T^+ = \{ \{c\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\} \}$$

Somit insgesamt:

$$\wp(M) = T^- \cup T^+ = \{ \varnothing, \{a\}, \{b\}, \{a,b\}, \{c\}, \{a,c\}, \{b,c\}, \{a,b,c\} \}$$

Ganz abstrakt ergibt sich somit der folgende rekursive Algorithmus:

Algorithmischer Ansatz 3.14 - Berechnung aller Teilmengen von M

Falls *M* leer, d.h. $M = \emptyset$:
leere Menge ist einzige Teilmengen

Falls *M* nicht leer: Wähle ein Element *a* aus *M* aus.
Berechne Sammlung T^- aller Teilmengen von *M* ohne *a* (rekursiv)
Bilde Sammlung T^+ aller Teilmengen, die *a* enthalten: Nimm dazu jede Menge aus T^- und bilde eine neue Menge, indem das gewählte a dazu genommen wird.
Die Menge aller Teilmengen ist die Vereinigung von T^- und T^+

Wenn Sie die rekursive Grundidee verstanden haben, sollte es kein Problem sein, die Umsetzung in Java-Code zu verstehen, siehe folgender Code (bzw. siehe Programmbeispiele in Moodle). `resultSet1` entspricht dabei T^-.

Implementierung 3.15 - Berechnung alle Teilmengen in Java

Berechnung alle Teilmengen der Werte im Teilarray `arr[0..maxIndex]`

```
private static <E> Set<Set<E>> allSubsets(E[] arr,
                                          int maxIndex) {
  Set<Set<E>> resultSet = new HashSet<Set<E>>();

  if (maxIndex >= 0) {
      // Menge ist nicht leer, wähle letztes Element im gegebenen Feldbereich
      E selected = arr[maxIndex];

      // bilde rekursiv alle Teilmengen ohne das gewählte Element selected
      Set<Set<E>> resultSet1 = allSubsets(arr, maxIndex - 1);
      // nimm jede dieser Mengen zum Ergebnis hinzu
      resultSet.addAll(resultSet1);

      // bilde alle Teilmengen, die selected enthalten
      for (Set<E> set1 : resultSet1) {
          // Erzeuge Kopie der Menge aus resultSet1 und nimm
          // gewähltes Element dazu
          Set<E> set2 = new HashSet<E>(set1);
          set2.add(selected);
          // füge die ergänzte Kopie zum Ergebnis hinzu
          resultSet.add(set2);
      }
  } else {
      // Menge ist leer. Leere Menge hat nur leere Menge als
      // einzige Teilmenge
```

```
        Set<E> emptySet = new HashSet<>();
        resultSet.add(emptySet);
    }

    return resultSet;
}
```

Um alle Teilmengen des kompletten Feldinhalts zu berechnen, brauchen wir nur noch folgende Wrapper-Methode zum Aufruf, die alle Teilmengen des gesamten Arrays berechnen lässt.

```
public static <E>  Set<Set<E>> allSubsets(E[] arr) {
    return allSubsets(arr, arr.length-1);
}
```

Aufgabe 3.16 - allSubsets testen

Sie finden die Implementierung in Moodle. Schauen Sie sich die Lösung an und berechnen Sie damit Teilmengen von unterschiedlich großen Mengen.

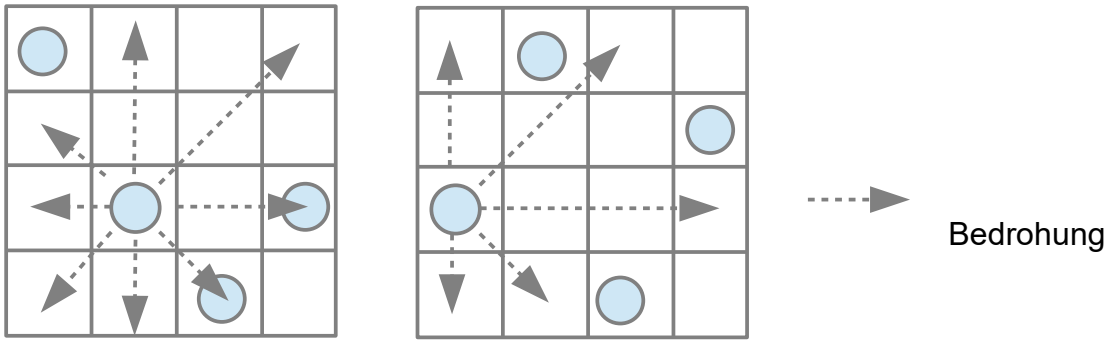
3.4 Algorithmenprinzip "Backtracking"

Eine weitere typische Anwendung der Rekursion sind Verfahren für die Suche nach der Lösung eines Problems, die dem Bereich der Künstlichen Intelligenz zugeordnet werden können. Ein grundlegendes Verfahren dafür ist sog. *Backtracking*. Das Verfahren soll an folgendem klassischen Knobelproblem erläutert werden.

Beispiel 3.17 - Das n-Damen-Problem

gegeben: Schachbrett der Seitenlänge *n*

gesucht: Positionen für *n* Damen, so dass sie sich gegenseitig nicht bedrohen. Eine Dame bedroht alle Figuren in der gleichen Spalte, der gleiche Reihe und auf beiden Diagonalen, jeweils in beide Richtungen und in beliebiger Entfernung.



n = 4

Das rechte Beispiel ist eine Lösung des *n*-Damen-Problems für *n* = 4.

Wie kann systematisch für gegebenes *n* nach einer Lösung des Problems gesucht werden? Gibt es eine Lösung für die übliche Schachbrettgröße *n* = 8?

3.4.1 Grundidee "Versuch-und-Irrtum" (trial and error)

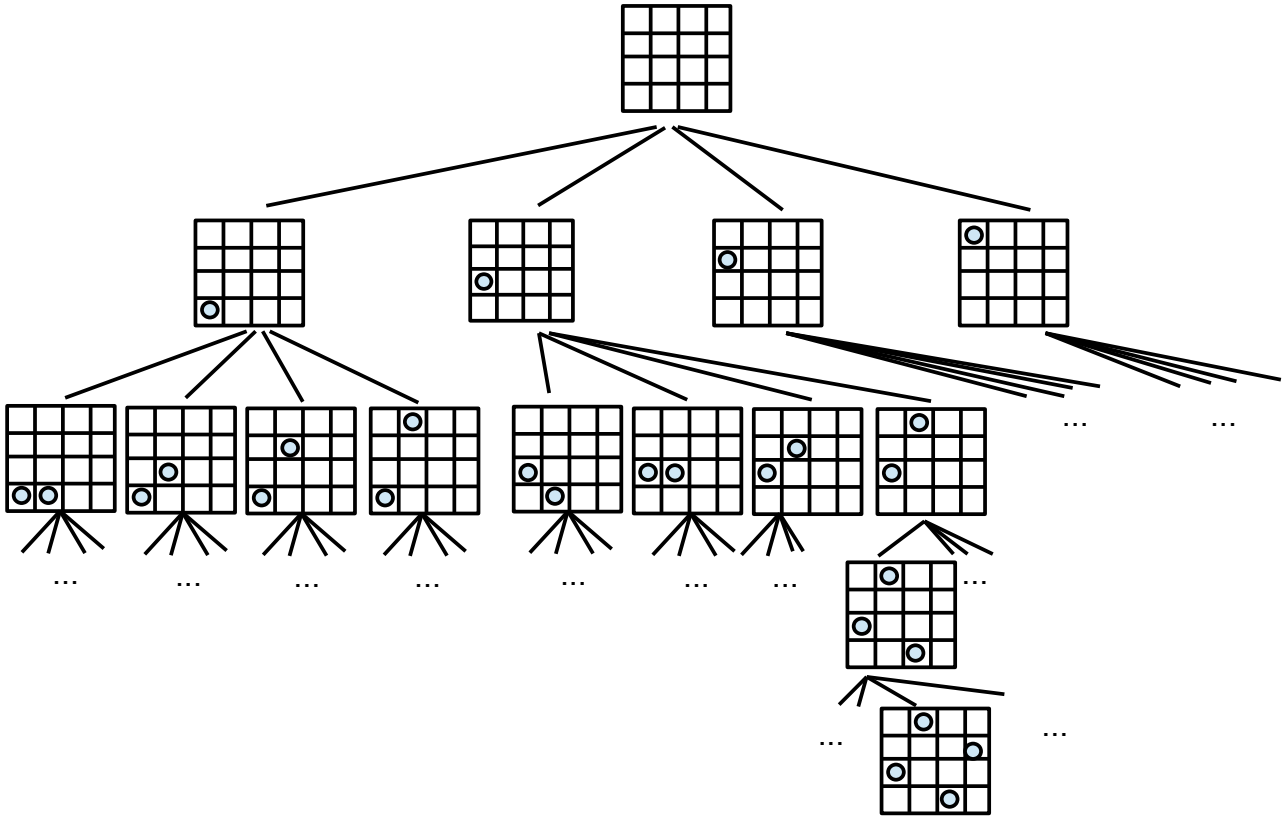
Für eine gegebene Problemstellung (wie z.B. *n*-Damen-Problem) soll ein Verfahren entwickelt werden, das nach einer Lösung sucht.

- Es wird vorausgesetzt, dass man eine Problemstellung hat, bei der Teillösungen schrittweise nach und nach erweitert und zu einer Gesamtlösung vervollständigt werden können.
- Kann eine neue Teillösung nicht zu einer endgültigen Lösung führen, wird der letzte Schritt bzw. werden die letzten Schritte zurückgenommen ("Backtracking") und es werden stattdessen alternative Möglichkeiten ausprobiert.

Wird bei der Lösungssuche Schritt für Schritt vorgegangen und hat man bei jedem Schritt mehrere Möglichkeiten, lassen sich alle Möglichkeiten insgesamt durch einen sog. Suchbaum darstellen.

Suchbaum für die Suche nach Lösungen

Nachfolgend ist ein Suchbaum für das 4-Damen-Problem skizziert. Es wird

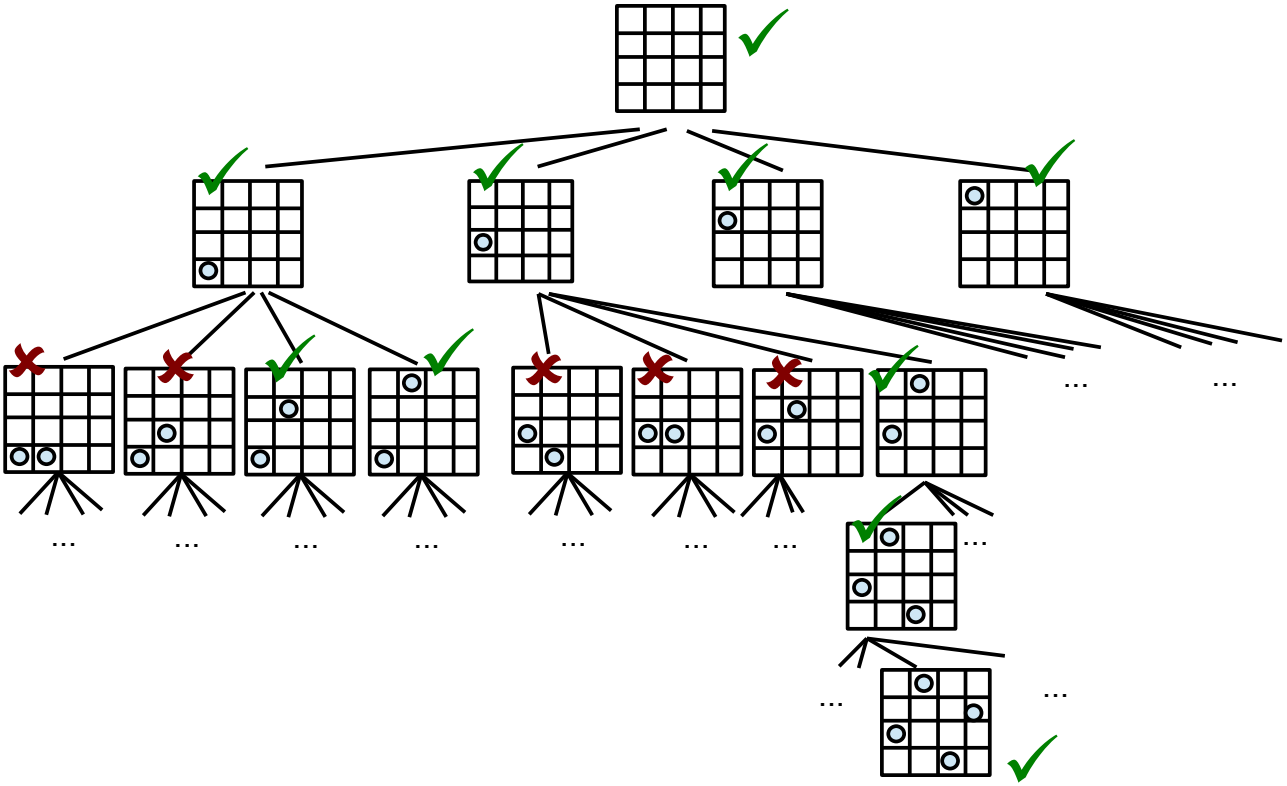


angenommen, dass spaltenweise von links nach rechts Damen gesetzt werden. Pro Spalte kann immer nur eine Dame gesetzt werden, da sonst eine Bedrohung vorliegen würde.

Der Suchbaum ist nur ein gedankliches Konzept, das die Suchmöglichkeiten darstellt, aber keine Datenstruktur, die tatsächlich aufgebaut oder gespeichert wird.

Bewertung der Knoten

Für jeden Knoten des Suchbaums, d.h. jede Brettbelegung, gibt es eine Bewertung, ob die Belegung zulässig ist oder zu einer Bedrohung führt.



Backtracking ist ein rekursives Verfahren für die "trail-and-error"-Vorgehensweise, mit dem die im Suchbaum vorhandenen Möglichkeiten systematisch abgesucht werden, bis eine Lösung gefunden wird.

Algorithmenprinzip 3.18 - Backtracking

boolean FindeLösung(*stufe*, *teilresultat*):

wiederhole, solange es noch unbehandelte Teillösungen auf dieser Stufe gibt:

wähle neue Teillösung auf Stufe *stufe*

falls Wahl für Teillösung gültig ist **dann**
erweitere *teilresultat* um neue Teillösung

```
falls teilresultat vollständige Lösung ist dann
        // fertig, Lösung gefunden!
        return true;
sonst
        //mit erweiterter Teillösung rekursiv weitersuchen
        falls FindeLösung(stufe+1, teilresultat) erfolgreich dann
                //Erfolgreiche Suche an Aufrufer zurückmelden
                return true;
        sonst:
                //Backtracking
                mache letzte Wahl auf aktueller Stufe rückgängig
//es gibt keine weiteren Teillösungen auf aktueller Stufe, Suche erfolglos
return false
```

Anwendung von Backtracking auf das n-Damen-Problem

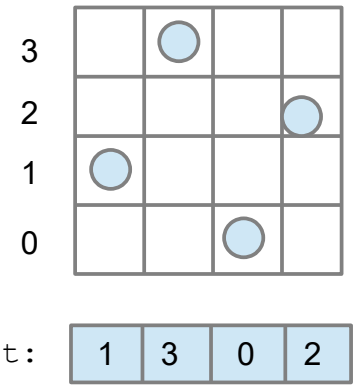
Beim Backtracking wird zwar nicht der Suchbaum gespeichert, aber der jeweils untersuchte aktuelle Teilzustand der Lösung muss irgendwie gespeichert werden.

Da es pro Spalte nur eine Dame geben kann, reicht ein eindimensionales Array

```
int[] brett
```

zur Repräsentation von (Teil-)Lösungen aus:

- pro Spalte eine Dame
- es wird im Array die Zeilenposition der Dame in dieser Spalte gespeichert

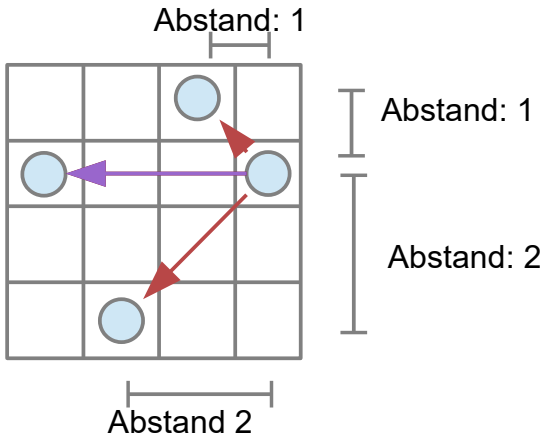


Wird eine neue Dame in der nächsten Spalte gesetzt, muss nur geprüft werden, ob dieses Dame die bisher schon gesetzten Damen bedroht. Bei den vorher gesetzten Damen wurde schon zuvor sichergestellt, dass sie sich gegenseitig nicht bedrohen.

Für eine neu gesetzte Dame muss geprüft werden, ob die Dame andere Damen bedroht:

- Dame in der gleichen Zeile wird bedroht. Dies ist daran zu erkennen, dass der Wert im Array gleich ist.

- Dame auf Diagonale wird bedroht. Die ist daran zu erkennen, dass Zeilenabstand = Spaltenabstand.



- In Java lässt sich das einfach formulieren:

Programm 3.19 - Prüfen, ob Bedrohung vorliegt

```
boolean istBedrohung(int[] brett, int neueSpalte) {
    for (int spalte = 0; spalte < neueSpalte; spalte++) {
        if (brett[spalte] == brett[neueSpalte] ||
            (neueSpalte-spalte) == Math.abs(brett[neueSpalte]
                                            - brett[spalte])) {
            return true;    //Bedrohung gefunden
        }
    }
    return false;           //keine Bedrohung gefunden
}
```

Damit kann nun die Backtracking-Suchmethode für das n-Damen-Problem geschrieben werden

Programm 3.20 - Rekursive Suchmethode mit Backtracking

```
boolean findeLösung(int[] brett, int spalte) {
    //Teillösungen für aktuelle Spalte durchgehen
    for(int zeile = 0; zeile < brett.length; zeile++) {
      //erweitere Lösung und neu gesetze Dame
        brett[spalte] = zeile;
      //prüfe, ob Erweiterung gültig ist, d.h. keine Bedrohung erzeugt
        if (!istBedrohung(brett, spalte)){
          //Wurde Geamtziel erreicht?
            if (spalte == brett.length-1) {
              //  alle Spalten besetzt, Gesamtlösung gefunden!
                return true;
```

```
            }
            // Lösung noch unvollständig, eine Spalte weitergehen
              spalte++;
              //rekursiv auf nächster Stufe weiter suchen
              if (findeLösung(brett, spalte)) {
                //Lösung gefunden, Erfolg zurückmelden
                    return true;
              } else {
                // Zweig im Suchbaum führte zu keiner Lösung
                // Backtracking: Erweiterung auf nächste Spalte
                // zurücknehmen
                    spalte--;
          }
      }
  // keine Teillösung in aktueller Spalte führte zum Erfolg
      return false;
}
```

Diese rekursive Backtracking-Methode muss nur noch in geeigneter Weise aufgerufen werden:

Programm 3.21 - Aufruf der Suchmethode für das n-Damen-Problem

```
void n_Damen(int n) {
    int[] brett = new int[n];

    if (findeLösung(brett, 0)) {
        System.out.println("Loesung für " + n + "Damen:");
        druckeBrett(brett);
    } else
        System.out.println("Keine Lösung gefunden");
}
```

Für $n = 8$ kann damit schnell eine Lösung des n-Damenproblems berechnet werden:

```
|D| | | | | | | |
| | | | | | |D| |
| | | | |D| | | |
| | | | | | | |D|
| |D| | | | | | |
| | | |D| | | | |
| | | | | |D| | |
| | |D| | | | | |
```

Anmerkungen

- Für größer werden Argumente n steigt der Berechnungsaufwand exponentiell.

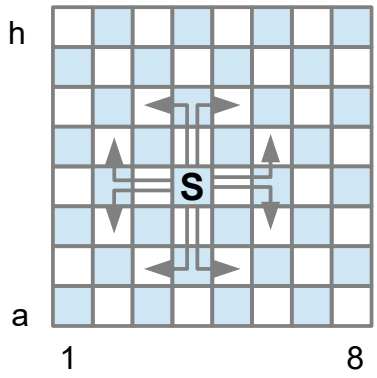
- Soll nicht nur eine Lösung gefunden werden, sondern möchte man alle Lösungen haben, muss der Backtracking-Algorithmus nur leicht modifiziert werden: Wurde eine Lösung gefunden, wird sie ausgegeben, danach wird dann aber nicht aufgehört, sondern einfach mit der Suche weitergemacht. (Eine Implementierung dafür finden Sie auch in Moodle bei den Programmbeispielen.)

Aufgabe 3.22 - Das Springerproblem

(eine echte Herausforderung!) Von Leonhard Euler, dem berühmten Mathematiker, wurde im Jahr 1758 folgendes Problem gestellt:

> *Gegeben sei ein leeres Schachbrett. Gibt es eine Zugfolge für den Springer, mit der er jedes Feld genau einmal besucht?*

Beim Schach hat ein Springer maximal 8 Zugmöglichkeiten, so wie hier rechts dargestellt ("zwei vor, eins seitwärts")



Schreiben Sie ein Programm, das mittels *Backtracking* dieses Problem löst, d.h. das auf einem Schachbrett der Seitenlänge *n* eine Folge von Zügen für einen Springer bestimmt, so dass jedes Feld des Schachbretts genau einmal besucht wird. Startpunkt soll die linke untere Ecke a1 sein (oder ggf. ein anderes Feld, das der Benutzer wählen kann).

[Sie werden vermutlich im Moment nicht die Zeit haben, das komplett zu programmieren. Sie sollten sich aber grob einen Lösungsansatz dafür überlegen.]

Hier ist eine Lösung für *n* = 5 mit Startpunkt a1 (unten links) zu sehen. Die Zahlen geben an, bei welchem Zug der Springer an dieser Position steht.

```
|21|16|11| 4|23|
|10| 5|22|17|12|
|15|20| 7|24| 3|
| 6| 9| 2|13|18|
| 1|14|19| 8|25|
```

Kap.4 Analyse von Algorithmen

Inhalt

- Korrektheit
- Terminierung
- Komplexität
- Asymptotisches Wachstum von Funktionen: O-, Ω-, Θ- Notation
- Rechnen in Größenordnungen
- Lineare und binäre Suche

4.1 Grundlegende Analyse von Algorithmen

In diesem Kapitel werden die methodischen Grundlagen dafür vorgestellt, wie zwei wesentliche Eigenschaften von Algorithmen analysiert werden können:

- **Korrektheit**: Funktioniert der Algorithmus richtig?
- **Komplexität**: Wie effizient ist der Algorithmus?

Werden Algorithmen entwickelt, sind immer beide Eigenschaften im Blick zu behalten. Wir werden uns im Lauf des Semesters vor allem mit der Effizienz von Algorithmen beschäftigen und wie man deren Komplexität beschreiben und analysieren kann.

An einem einfachen einführenden Beispiel betrachten wird diese beiden Aspekte.

Aufgabe 4.1 - akkumulieren

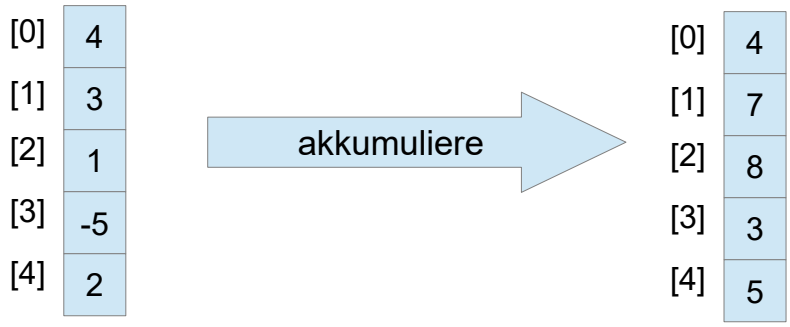
Es soll eine Methode

```
public static void akkumuliere(double[] daten)
```

implementiert werden, die in folgender Weise die Werte des Arrays `daten` aufsummiert:

$$daten_{neu}[i] = \sum_{k=0}^{i} daten_{alt}[k]$$

d.h. der *n*-te Eintrag im Feld soll danach die Summe der ersten *n* Werte sein, so wie in der folgenden Abbildung beispielhaft dargestellt.



Die Methode soll auch für sehr große Felder (> 1 Mio. Elemente) gut geeignet sein. Die Ergebnisse sollen im gleiche Array zurückgegeben werden.

Es werden nun drei Lösungsvorschläge dafür vorgestellt.

- **Lösungsvorschlag 1**: Jede Position des Arrays wird in der äußeren Schleife durchgegangen und in der inneren Schleife werden die Werte bis zu dieser Position aufsummiert (in Anlehnung an die Summenformel s.o.)

```
public static void akkumuliere_1(double[] daten) {
  for (int i = 0; i < daten.length; i++) {
      double summe = 0;
      for (int k = 0; k <= i; k++) {
          summe += daten[k];
      }
      daten[i] = summe;
  }
}
```

- **Lösungsvorschlag 2**: ähnlich wie Lösungsvorschlag 1, allerdings wird das Array von hinten nach vorne durchgegangen, um die entsprechende Summe zu bilden.

```
public static void akkumuliere_2(double[] daten) {
  for (int i = daten.length - 1; i >= 0; i--) {
      double summe = 0;
      for (int k = 0; k <= i; k++) {
          summe += daten[k];
      }
      daten[i] = summe;
  }
}
```

- **Lösungsvorschlag 3**: Diese Lösung arbeitet mit einem einzigen Schleifendurchgang.

```
public static void akkumuliere_3(double[] daten) {
  for (int i = 1; i < daten.length; i++) {
      daten[i] = daten[i-1] + daten[i];
  }
```

```
}
```

Sind diese Vorschläge *korrekt* und sind sie *effizient*?

	korrekt?	effizient?
Lösungsvorschlag 1		
Lösungsvorschlag 2		
Lösungsvorschlag 3		

Aufgabe 4.2 - Laufzeitmessung für `akkumuliere`

Sie finden in Moodle den Programmcode in Klasse `Summe`. Messen Sie mit Methode `test2` die Laufzeit von Version 2 und 3 für Feldgrößen n = 1 000, 10 000, 100 000, 1 000 000, 10 000 000) – sofern möglich.

Verwenden Sie für Laufzeitmessungen im Rahmen dieser Vorlesung für den Java-Compiler immer die Option -Xint, die den Just-in-time-Compiler abschaltet.

Sind signifikante Unterschiede zu erkennen?

4.2 Korrektheit

Der Schwerpunkt dieses Semesters liegt auf der Effizienz von Programmen. Die Korrektheit spielt trotzdem eine entscheidende Rolle: Sofern ein Programm nicht korrekt funktioniert, nützt es auch nichts, wenn es effizient ist. Bevor man sich Gedanken um die Effizienz eines Programms macht, muss man sich schon Gedanken zur Korrektheit des Programms gemacht haben.

In diesem Abschnitt geht es deshalb um den Begriff der Korrektheit. Was heißt überhaupt, dass ein Programm korrekt ist? Wie kann man „beweisen“, d.h. sich genau überlegen, dass ein Programm korrekt ist, insbesondere bei üblichen Programmen mit Schleifen (bei Rekursion haben Sie mit der Berechnungsinduktion schon eine Methode kennengelernt)?

Als Beispiel betrachten wir u.a. Sortieralgorithmen. *Insertionsort*, ein einfaches Sortierverfahren, haben sie vermutlich auch schon in Programmieren kennengelernt:

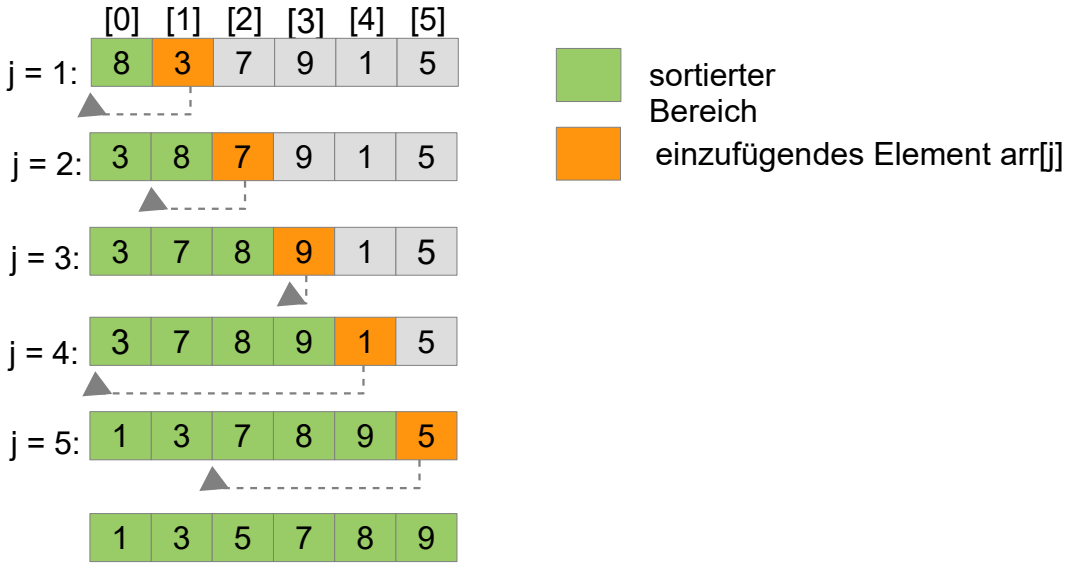
Algorithmische Grundidee von Insertionsort

- Am Anfang des Arrays wird ein sortierter Bereich aufgebaut, in den nach und nach die folgenden Elemente *eingefügt* werden (deshalb "*Insertion*sort").
- Am Anfang besteht der sortierte Bereich nur aus dem ersten Element arr[0].
- In jedem Durchgang wird das nächste Element an passender Stelle in den schon sortierten Bereich eingefügt, indem es so weit nach vorne geschoben wird, bis entweder der Anfang des Arrays erreicht ist oder das Element davor

kleiner oder gleich ist (Zeile 9-14). Die größeren Werte des sortierten Bereichs müssen dabei um eine Position nach hinten gerückt werden (Zeile 10). Dadurch wird der sortierte Bereich pro Durchgang der äußeren Schleife um ein Element vergrößert

- Wenn alle Elemente eingefügt wurden, ist das komplette Feld sortiert.

Beispiel 4.3 – Insertionsort



Algorithmus 4.4 - Insertionsort (Sortieren durch Einfügen)

```
1  static void insertionSort(double[] arr) {
2    int j = 1;
3    while (j < arr.length) {
4        //Element arr[j] an richtiger Stelle in den sortierten Bereich a[0] bis a[j-1]
5        //einfügen
6        double value = arr[j];       //einzufügendes Element
7        // richtige Stelle zum Einfügen suchen
8        int insertPos = j;
9        while(insertPos > 0 && arr[insertPos-1] > value) {
10          arr[insertPos] = arr[insertPos-1];
11          insertPos--;
12       }
13       arr[insertPos] = value; //Wert an richtiger Position einfügen
14       j++;
15    }
16 }
```

Aufgabe 4.5 - Korrektheit von Insertionsort

- Wie kann begründet werden, dass Insertionsort richtig sortiert?

Bevor diese Frage beantwortet werden kann, muss erst einmal geklärt werden, was es heißt, korrekt zur sortieren.

Definition 4.6 - Korrektheit von für Sortierverfahren

Enthält ein Array *arr* am Anfang die *n* Elemente $[a_1, a_2, \ldots, a_n]$, dann muss am Ende nach dem Sortieren gelten:

- **Permutation**: Die Elemente $[a'_1, a'_2, \ldots, a'_n]$ von *arr* sind eine Permutation (Umordnung) der ursprünglichen Elemente $[a_1, a_2, \ldots, a_n]$.
- **Monotonie**: Für alle Indizes *i* von 1 bis *n*-1 gilt:

  $a'_i \leq a'_{i+1}$ (bei aufsteigender Sortierung)

  bzw.

  $a'_i \geq a'_{i+1}$ (bei absteigender Sortierung)

Anmerkungen

- Wenn nicht anderes angegeben wird, ist mit "sortiert" zukünftig (auch in der Klausur) immer "aufsteigend sortiert" gemeint.
- Enthält ein Array mehrfach den gleichen Wert, so bleiben diese Werte beim Sortieren auch mehrfach erhalten.

Fazit zu Lektion 4

Das haben Sie in dieser Lektion gelernt

- Wie lassen sich rekursiv alle Teilmengen einer Menge berechnen?
- Was versteht man unter Backtracking? Wozu wird es eingesetzt?
- Wie wird Backtracking programmiert?
- Was ist ein Suchbaum?
- Welche zwei grundlegenden Eigenschaften sind für die Analyse von Algorithmen wichtig.
- Welcher Zusammenhang besteht zwischen Korrektheit und Komplexität von Algorithmen